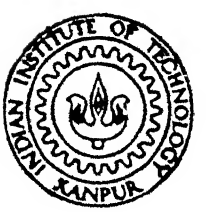
GENERATION OF COMBINATORIAL OBJECTS ON SERIAL/PARALLEL COMPUTERS

by

ATUL SIBAL



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

FEBRUARY, 1988

GENERATION OF COMBINATORIAL OBJECTS ON SERIAL/PARALLEL COMPUTERS

A Thesis Submitted

In Partial Fulfilment of the Requirements

for the Degree of

MASTER OF TECHNOLOGY

by

ATUL SIBAL

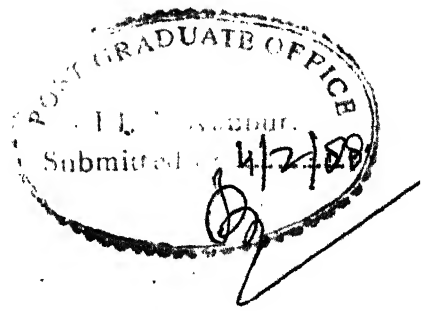
to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

FEBRUARY, 1988

CSE-1988-M-SIB-GEN



CERTIFICATE

This is to certify that the thesis entitled Generation of Combinatorial Objects on Serial/Parallel Computers is a report of work carried out under our supervision by ATUL SIBAL and that has not been submitted elsewhere for a degree.

R.K. Ghosh

Dr.R.K. Ghosh
Asst. Professor
Dept. of C.S.E.
I.I.T., Kanpur.

Phalguni Gupta

Dr.P. Gupta
Asst. Professor
Dept. of C.S.E.
I.I.T., Kanpur.

Place: Kanpur

Date : February 4, 1988.

ACKNOWLEDGEMENTS

It gives me great pleasure to put on record my feeling of indebtedness to Dr. R. K. Ghosh and Dr. P. Gupta, my thesis supervisors, for creating an atmosphere of freedom in which it was a pleasure to work. Inspite of their busy schedules they could spare enough time to give me regular guidance and ensure smooth progress.

I am also grateful to all my classmates and residents of Hall IV, IIT Kanpur for making my stay at IIT-K a memorable one.

ATUL SIBAL

ABSTRACT

The present work deals with parallel algorithms for problems related to generation of combinatorial objects on the SIMD (SM-CREW) computer. Parallel algorithms for the problems of generation of a Random Sample, generation of Binary trees, and generation of Random Rooted Unlabeled tree have been presented. In addition, sequential algorithms for generating a Random Binary tree have been proposed.

The algorithms for generating a random sample of k out of n items achieve optimal speedup ratio. They run in O(log k) time using O(k) processors and O(k) memory. The random sampling algorithm has been used to generate a random permutation of n objects in $O((\log n)^2)$ time using O(n) processors.

The random unlabeled rooted tree is found in O(log k) time using O(k) processors.

Two algorithms have been presented for lexicographic generation of binary trees with n nodes. The first algorithm takes O(BT(n)) time with O(n) processors while the second takes $O(\lceil BT(n)/P \rceil \cdot n)$ time, where P is the number of processors and BT(n) is the total number of binary trees with n nodes.

The sequential algorithms presented for generating a random binary tree take optimal O(n) time using O(n) space.

CONTENTS

		Page
	ACKNOWLEDGEMENTS	i
	ABSTRACT	ii
	CONTENTS	iii
1	INTRODUCTION	
	1.1 Parallel Computing and Combinatorial Algorithms.	1
	1.2 Model of Computation	3
	1.3 Language Constructs	4
	1.4 An Overview of the Thesis	8
2	GENERATION OF RANDOM SAMPLE	
	2.1 Introduction	1Ø
	2.2 An Algorithm for Parallel Random Sampling	13
	2.3 Another approach to Parallel Random Sampling	24
	2.4 Random Permutation in Parallel	37
	2.5 Conclusion	4Ø
3	GENERATION OF RANDOM UNLABELED ROOTED TREE & RANDOM BINARY TREE	
	3.1 Introduction	41
	3.2 Parallel Algorithm for Generation of Random Unlabeled Rooted tree	42
	3.3 Random Binary Tree Generation	48
	3.4 Conclusion	56
4	GENERATION OF BINARY TREES	
	4.1 Introduction	57
	4.2 Definitions and Theoretical Background	57
	4.3 Generation Algorithms	63
	4.4 Conclusion	72
	REFERENCES	74

CHAPTER 1

INTRODUCTION

1.1 PARALLEL COMPUTING & COMBINATORIAL ALGORITHMS

There is a fundamental limit imposed to the computing power of a sequential computer due to what is now famous as the 'speed of light' argument. The speed of light in vaccum is 3×10^8 m/s and 3×10^7 m/s in silicon. This imposes a limit to the number of floating point operations per second which can be done on a sequential machine. As the size of the problems increases, the power of whatever sequential machine is available is exhausted and hence power beyond the ability of a single processor machine is required. This makes parallel processing inevitable for large scale problems. Also, in a real-time environment parallel processing is required to reduce the turn-around time which is critical to the performance of the system.

Several commercial parallel systems are available now like INTEL's iPSC, Thinking Machines' Connection Machine, FPS hypercube supercomputer, and the BBN Butterfly system. However, what we still do not know is how to program these systems efficiently. We are good at understanding sequential algorithms, but we have little experience with algorithms for parallel processors. Though the area of Parallel Algorithms has generated considerable research activity, a lot

more needs to be done before we can really understand the mechanics of algorithm design for parallel machines. The present work, which presents some parallel combinatorial algorithms, is a small step in that direction.

The field of *combinatorial algorithms* deals with the problems of performing computations on discrete, finite mathematical structures. The practical importance of computations which are combinatorial in nature has contributed to the considerable research activity in this area. Sorting, searching, generation of combinatorial objects, and graph algorithms are some of the extensively studied areas of combinatorial computing [RND78].

Following FLYNN's classification scheme [FL66], parallel computers are broadly classified into SIMD (Single Instruction Stream Multiple Data Stream) computers and MIMD (multiple instruction stream Multiple Data Stream) computers. In SIMD computers, a stream of instructions is executed by a number of synchronised processors, each operating upon its own memory. In MIMD computers each processor has an independent instruction counter and operates in a speed independent manner. Most of the research in Parallel Algorithms has been done for the SIMD model. For some major results in the area of parallel algorithms see [KP81],[AS85],[QD84].

A good deal of work has been reported on parallel combinatorial algorithms, on the Shared memory model of the SIMD computer (see section 1.3). Several algorithms for

parallel generation of combinatorial objects have been proposed. MOR & FRAENKEL [MF82], CHEN & CHERN [CC86] and GUPTA & BHATTACHARJEE [GB82],[GB83] have dealt with the problem of parallel generation of Permutations. Algorithms for parallel generation of Combinations have been proposed by GUPTA & BHATTACHARJEE [GB81], CHEN & CHERN [CC86] and CHAN & AKL [CA86].

1.2 MODEL OF COMPUTATION

A shared memory model of Single Instruction-stream Multiple Data-stream (SIMD) computer is used as the model of computation for the parallel algorithms described in the subsequent chapters of this thesis. This model has been used by many authors for a wide variety of problems.

In this model there are a master processor and a number of slave processors. It is assumed that the slave processors have a limited amount of local memory and there is an unbounded global memory which these slave processors can access. The master processor broadcasts instructions to the slave processors. The slave processors in active or enable mode execute the instruction broadcast by the master processor at the same time. It is further assumed that all the processors are synchronized in the sense that if a set of instructions is executed in parallel, then each must be allowed to finish before the next set of instruction is started.

In this model no two slave processors are allowed to

write on the same location of the global memory, however, two or more slave processors can read from the same location of the global memory. This model is known as a **Parallel Random Access Machine (PRAM)** which allows **Concurrent Read** and **Exclusive Write (CREW)**.

For measuring time complexity of an algorithm designed for this model, it is assumed that the cost of one execution of an instruction broadcast by the master processor is one unit time, irrespective of the number of slave processors that may be active. The requirement of the processors is determined on the basis of the maximum number of the slave processors active at an instant during the execution of the entire algorithm.

1.3 LANGUAGE CONSTRUCTS

The following constructs have been used to present the algorithms in this thesis:

The parallel loop involving n processors is represented by the following statement :

```
for i := p to p+n-1 pardo

        steps to be executed
        in parallel;

dopar;
```

Here p is the staring index of the active processors. Similarly, the statements to be executed in parallel can be

denoted by:

```
parbegin
    statements to be executed
    in parallel
parend;
```

Similarly, sequential begin and sequential end are represented by **begin** and **end** respectively.

An algorithm is presented in the form of a procedure with appropriate declarations. The prefix **par** in a procedure name indicates that it refers to a parallel algorithm, otherwise it is a sequential algorithm. Besides input and output parameters, the argument list of parallel procedures includes the starting index of the set of processors and the total number of active processors. A typical example of a procedure statement is provided below.

```
proc par_name (p,q; input; output);
```

where;
- **p** : starting index of the set of active processors.
- **q** : total number of active processors
- **input** : list of input variables separated by commas.
- **output** : list of output variables separated by commas.

Let us consider some examples of parallel procedures to illustrate the above constructs.

The first example considers the following problem: Given n integers in an array A[1:n] generate the following n sums:

$$SUMA[i] := \sum_{j=1}^{i} A[j], \quad 1 \leq i \leq n.$$

The algorithm proceeds in $\lceil \log n \rceil$ steps as follows:

```
proc  par_find_parsum  (p,n; A,n; SUMA);

var
  /* input */
  n : integer;
  A : array [1..n] of integer;
  /* output */
  SUMA : array [1..n] of integer;
  /* local */
  i,u : integer;
  FAR : array [1..n] of 1..n;
begin
I1: /* initialize */
    for i:= p to p+n-1 pardo
       begin
          u := i-p+1;
          FAR[u] := u+1;
          SUMA[u] := A[u]
       end
    dopar;
    FAR[n] := n;
I2: /* Do folding operations on the FAR pointer for
       ⌈log n⌉ steps. */
    for ⌈log n⌉ steps do
       for i := p to p+n-1 pardo
          begin
             u := i-p+1;
             if u <> FAR[u] then
                SUMA[FAR[u]] := SUMA[FAR[u]] + SUMA[u];
             FAR[u] := FAR[FAR[u]]
          end
       dopar
end;
```

LEMMA 1.1

The procedure par_find_parsum runs in O(log n) time with O(n) processors.

proof: see ref. [AS86] []

The next example addresses the following problem: Given a set of k integers in the range 1 to n (with some integers having repeated occurences) in an array SAMPLE[1:k], find the set of distinct integers present in SAMPLE[1:k]. If 'items' is the number of distinct integers, then SAMPLE[1:items] stores the output. The algorithm proceeds by replacing all repeated occurences of any integer by n+1 (which is outside the range of the integers) and then sorting the array SAMPLE. All the distinct integers are thus collected to one end of the array. The procedure **par_compress** presents the parallel implementation of the algorithm.

```
proc  par_compress
( p,k ; SAMPLE,k,n ; SAMPLE,items );

var
   /* input */
   SAMPLE : array [1..k] of 1..n;
   /* output */
   SAMPLE : array [1..items] of 1..n;
   items : integer;
   /* local */
   i,u : integer
```

```
begin
I1: /* sort the array SAMPLE */
    par_sort (SAMPLE[1:k]);
I2: /* set the duplicate items in SAMPLE to n+1. Note
       that the range of items in SAMPLE is 1..n */
    for i := p to p+k-2 pardo
       begin
          u := i-p+1;
          if SAMPLE[u] = SAMPLE[u+1] then SAMPLE[u+1] := n+1
       end
    dopar;
I3: /* again sort SAMPLE */
    par_sort (SAMPLE[1:k]);
I4: /* find the value of the last index 'items' */
    for i := p to p+k-2 pardo
       begin
          u := i-p+1;
          if (SAMPLE[u] <> n+1) and (SAMPLE[u+1] = n+1) then
             items := u
       end
    dopar
end;
```

Here the procedure **par_sort** is the parallel sorting procedure [CO86].

LEMMA 1.2 The procedure par_compress runs in O(log k) time with O(k) processors.

1.4 AN OVERVIEW OF THE THESIS

In the present work parallel algorithms have been presented for some problems in generation of combinatorial objects on the Shared memory model of the SIMD computer.

Chapter 2 presents two parallel algorithms for generating a **random sample** of k out of n objects in O(log k) time using O(k) processors and O(k) memory. A random permutation of n objects is then generated in $O((\log n)^2)$ time using O(n) processors.

Chapter 3 presents a parallel algorithm for generating a **random unlabeled rooted tree** with n nodes and k levels in O(log k) time using O(n) processors. Also presented are two sequential algorithms for the problem of generating a **random binary tree** with n nodes in O(n) time using O(n) space.

In chapter 4, parallel algorithms have been developed for **generation of all binary trees** with n nodes, lexicographically. If BT(n) is the total number of such trees, the first algorithm takes O(BT(n)) time with O(n) processors. The second algorithm takes $O(\lceil BT(n)/P \rceil . n)$ time, where P is the number of processors used.

CHAPTER 2
GENERATION OF RANDOM SAMPLE

2.1. INTRODUCTION

The problem of randomly selecting k items out of n distinct items, without replacement, arises in various applications like Quality Control, Market surveys, Simulation Studies etc. Without loss of generality assume that the n items are indexed 1 through n. So the above problem of drawing a random sample of size k out of n items is reduced to the problem of generating the indices of k items at random, where each index is a positive integer not greater than n.

The above problem is rather simple to solve if one allows either the memory storage used or the time complexity (or the number of processors used in the case of parallel computation) to be a function of n [NW75]. But in most practical situations where random sampling may be used, n is very large as compared to k. For example, consider the problem of selecting 3 distinct integers from the set {1,..,10000}, in which case we would like the time, space or processor complexities of the algorithm to depend on '3' rather than on '10000'.

Several algorithms exist for the problem of random sampling for sequential computers. These algorithms can broadly be classified into two categories. In the first category are

in a linear order, i.e. in the increasing order of their indices, on-line [VI84],[AD85].

The general form of algorithms in this category is as follows:

step1. Generate a random integer d.

step2. Skip the next d items and select the following one for the sample

N := N - d - 1

k := k - 1

if k > Ø go to step1.

All the algorithms differ mainly in how the skip distance 'd' is generated in step1. It may follow a uniform, geometric, or some other distribution. The best known algorithm in this category runs in O(k) time on the average ,but only O(n) time in the worst case, using O(k) space. A comparison of the major algorithms in this category has been given in TABLE 1.

NAME of ALGORITHM	REFERENCE	AVERAGE TIME COMPLEXITY	WORST CASE TIME COMPLEXITY	SPACE COMPLEXITY
S	[FMR62], [JO62]	O(n)	O(n)	O(k)
A	[VI84]	O(n)	O(n)	O(k)
D	[VI84]	O(k)	O(n)	O(k)
SG, SH, SG*	[AD85]	O(k)	O(n)	O(k)

TABLE 1.

The algorithms in the second category generate the sample in an arbitrary order. The sample may or may not be generated on-line. An algorithm in this category is the algorithm **select** due to GOODMAN and HEDETNIEMI [GH77] which takes O(n) time and O(n) space. Some of the more efficient algorithms in this category are discussed in section 2.3.

The best known sequential algorithm for random sampling which uses optimal O(k) memory [GUP] runs in O(k.log k) time in the worst case. Thus it is interesting to design a parallel algorithm whose product of time and processor complexities is O(k.log k) and which uses O(k) space.

In this chapter, two parallel algorithms (**par_random_sample** and **par_random_sample2**) for the problem of 'Random Sampling' are presented. The algorithms run on the Parallel Random Access Machine (PRAM) which allows for Concurrent Read and Exclusive Write (CREW) in a shared memory environment. Both the algorithms take O(log k) time using O(k) processors and O(k) memory. Further, the random sampling algorithm has been used to get a random permutation of n items in $O((\log n)^2)$ time using O(n) processors.

Section 2.2 and 2.3 present the algorithms **par_random_sample** and **par_random_sample2** while the random permutation algorithm is presented in section 2.4.

2.2. AN ALGORITHM FOR PARALLEL RANDOM SAMPLING

The algorithm **par_random_sample** presented in this section generates a random sample of k out of n distinct items without replacement. The algorithm considers the case when $n \leq k^2$ separately from the case when $n > k^2$.

CASE 1 $n \leq k^2$

In this case the problem of selecting the k items can be redefined as follows: Select k_l items from the first n/2 items and k_r items from the last n/2, where k_l is chosen at random in the interval [Ø,k], and $k_l + k_r = k$. Thus the k items can be selected recursively. The recursion ends when the number of items to be selected equals the number of items present in the corresponding group of items, or when the number of items to be selected equals 1. In case n is odd at any stage, one item (with index 'temp') which is selected randomly is rejected before starting the above procedure. This step is implemented by, first selecting the sample out of the first n-1 items, and later, incrementing all those items which are greater than or equal to 'temp' by one. The procedure **par_rsamp_case1** implements case 1 of the algorithm. The function **RAND[1,n]** computes a random integer in the range 1 through n.

```
proc  par_rsamp_case1  (p,k ; n,k ; SAMPLE);
var
  /* input */
  k,n : integer;
  /* output */
  SAMPLE : array [1..k] of 1..n;
  /* local */
  i,temp,flag : integer
begin
  if k = n then
     for i := p to p+k-1 pardo
        begin
           u := i-p+1;
           SAMPLE[i] := i
        end
     dopar
  else
     if k = 1 then
        SAMPLE[1] := RAND[1,n]
     else
        begin
           if n is odd then
              begin
                 temp := RAND[1,n];
                 n := n-1;
                 flag := 1
              end;
           begin
              Kl := RAND [Ø,min(k,n/2)];
              Kr := k - Kl;
              parbegin
                if Kl > Ø then
                  par_rsamp_case1(p,Kl;n/2,Kl;SAMPLE[1:Kl]);
                if Kr > Ø then
                  par_rsamp_case1(p,Kr;n/2,Kr;SAMPLE[Kl+1,Kr
              parend
           end
           if flag = 1 then
              for i := p to p+k-1 pardo
                 begin
                    u := i-p+1;
                    if SAMPLE[u] ≥ temp then
                       SAMPLE[u] := SAMPLE[u] +1
                 end
              dopar
        end
end;
```

LEMMA 2.1

For $n \leq k^2$ the procedure **par_rsamp_case1** computes a random sample of size k out of n in O(log k) time using O(k)

processors and O(k) memory.

proof:

The time complexity of the algorithm is given by

$T(n,k) := T(n/2,k) + O(1)$

or, $T(n,k) := O(\log n)$

For $k \leq n \leq k^2$, $O(\log n) = O(\log k)$. Since at any level of recursion we are selecting at most k items, the processor complexity of the algorithm is O(k). By symmetry, it is obvious that the sample selected by the above procedure is random. (At every stage k is broken down at random into k_l and k_r, and thus the probability of generating the partition (k_l,k_r) is the same as that for the partition (k_r,k_l).) The memory requirements of the algorithm are in the form of the global array SAMPLE[1:k] and the temporary variables 'temp','flag' which take O(1) memory for each recursive call to the procedure. There may be at most O(k) recursive calls and hence, the total memory requirement is O(k).

[]

CASE 2 $n > k^2$

In case $n > k^2$, the n items are subdivided into $\lceil n/k \rceil$ groups, each of size k (except one). Instead of directly choosing a sample of k items, a group number lying in the interval $[1, \lceil n/k \rceil]$ is generated at random for each of the k items to be selected for the sample. Let the group number i be generated by k_i random variates implying that k_i items are selected from the i^{th} group. The following observations

can be made:

1. $\sum k_i := k$

2. The number of groups selected are at most k (since only k group numbers are selected).

Case 2 is further subdivided into Case 2.1 and Case 2.2 depending upon whether n is a multiple of k or not.

CASE 2.1 n is a multiple of k ($n = \lceil n/k \rceil . k$)

In this case it is ensured that the size of all groups is equal to k. Since only k group numbers are generated, in the worst case all the group numbers generated are equal to some group i. Thus $k_i = k$, whereas $k_j = \emptyset$, $j<>i$. Hence the LEMMA 2.2 follows:

LEMMA 2.2

For any group i, the number of times it is selected (k_i) does not exceed the the number of items!preseot in it (= k) ,when $n = \lceil n/k \rceil . k$.

[]

LEMMA 2.3

If $\lceil n/k \rceil > k$ then there exists a contiguous block of at least k-1 items from which no item is selected for the sample (Assuming circularity of items i.e. the item number 1 follows item number n).

proof:

Since we have assumed that $\lceil n/k \rceil > k$ or $n > k^2$, in the

worst case, all the items selected for the sample are evenly distributed over the population. The minimum value of n is $k^2 + 1$, in which case we select every k^{th} item and thus, between any two items selected, there exists a gap of at least k-1.

[]

Knowing k_j for each j with $k_j <> \emptyset$, we can break up the original problem into subproblems for each such group j, i.e. at the next stage of recursion we select k_j items from group number j (having k items). The recursion ends when k_j becomes greater than square root of the size of group j, in which case the algorithm **par_rsamp_case1** is invoked.

CASE 2.2 n is not a multiple of k (n <> $\lceil n/k \rceil . k$)

In this case, if the n items are subdivided into $\lceil n/k \rceil$ groups, one group will have $(n - \lfloor n/k \rfloor . k)$ items (at most k-1) while the remaining $\lfloor n/k \rfloor$ groups have k items each. However, from LEMMA 2.3, we know that there exists a contiguous block of size at least k-1 from which no item is selected for the sample. So, a random variate 'breakpt' in the interval [1,n] is generated, and $[n-\lfloor n/k \rfloor . k]$ items starting from that point onwards are rejected to start with. Now, the total number of items available is a multiple of k ,and thus the case is same as case 2.1. The same technique can be applied at subsequent steps of recursion. The rejection of the $[n-\lfloor n/k \rfloor . k]$ items can be done by, first storing the pointer 'breakpt', then selecting the sample from the first

$\lfloor n/k \rfloor . k$ items, and later modifying those sample items which are greater than or equal to 'breakpt'. The modification can simply be done by adding $[n - \lfloor n/k \rfloor . k]$ to each of affected items. If 'breakpt' is greater than $\lfloor n/k \rfloor . k$, based on the assumption of circularity of items, all the sample items are modified.

Based on the above discussion we have the following major steps involved in selecting k out of n items when $n > k^2$ (as presented in procedure **par_rsamp_case2**) :

1. After dividing n into $\lceil n/k \rceil$ groups and generating k group numbers randomly, finding the group numbers selected (GROUP[1:T]) and the number of times each one is selected (KI[1:T]). (procedure **par_findki**).
2. After solving the random sampling problem for each of the groups selected, modifying the sample generated to take care of case 2.2 (procedure **par_modify_sample**).

```
proc  par_findki  ( p,k ; n,k,GROUP ; GROUP,KI,T );
var
   /* input */
   n,k : integer;
   /* output */
   T : integer;
   GROUP : array [1..T] of 1..⌈n/k⌉;
   KI : array [1..T] of 1..k;
   /* local */
   STAGE : array [1..k] of 1..k;
   u,i : integer
```

```
begin
I1: par_SORT (GROUP[1:k]);
I2: for i := p to p+k-2 pardo
       begin
          u := i-p+1;
          if GROUP[u] = GROUP[u+1] then
             GROUP[u+1] := n+1;
          STAGE[u] := u
       end
    dopar;
I3: Sort the set A = {(GROUP[i],STAGE[i]) | i:= 1,..,k}
    such that for i<j, GROUP[i] < GROUP[j];
I4: for i:= p to p+k-1 pardo
       begin
          u := i-p+1;
          if (GROUP[u] <> n+1) and (GROUP[u+1] = n+1) then
             T := i
       end
    dopar;
I5: for i:= p to p+T-1 pardo
       begin
          u := i-p+1;
          KI[u] := STAGE[u+1] - STAGE[u]
       end
    dopar
end;
```

LEMMA 2.4

The procedure **par_findki** is correct.

proof:

The array GROUP[1:k] which is the input to this procedure, stores the generated group numbers. Step I1 sorts the array GROUP[1:k]. Steps I2-I5 find, for each group selected, the number of times it is selected. This is done by manipulating a set A = {(GROUP[i],STAGE[i])|i:=1,..k}. In step I2 STAGE[i] is initialized to i and every repeated occurence of any group number in GROUP[1:k] is replaced by n+1 (n+1 is not a valid group number). Let T be the number of distinct group numbers. The steps I3,I4 collect all the distinct group numbers into GROUP[1:T]. Step I5 computes the number of times each group is selected. If KI[u] is the

number of times GROUP[u] is selected, it can easily be obtained by taking the difference of STAGE[u+1] and STAGE[u], where STAGE[u] gives the starting index of GROUP[u] for u:=1,..T.

[]

```
proc par_modify_sample (p,T; k,n,T,breakpt,SAMPLE; SAMPLE);
var
   /* input */
   k,n,T,breakpt : integer;
   SAMPLE : array [1..k] of 1..n;
   /* output */
   SAMPLE : array [1..k] of 1..n;
   /* local */
   i,u,v,m1,m2,st,lt,rt,asize,incr,remainder : integer;
begin
I1: asize := ⌈n/k⌉.k;
I2: remainder := n-asize;
I3: if breakpt < asize then
       for i := p to p+k-1 pardo
          begin
             u := i-p+1;
             if SAMPLE[u] > breakpt then
                SAMPLE[u] := SAMPLE[u] + remainder
          end
       dopar
    else
       begin
          incr := breakpt - asize + 1;
          for i := 1 to k pardo
             begin
                u := i-p+1;
                SAMPLE[u] := SAMPLE[u] + incr
             end
          dopar
       end
end;
```

The procedure **par_rsamp_case2** which presents the algorithm for case 2 calls the procedure **par_find_parsum** of chapter 1 to compute, for 1<i<T, SUMKI[i] = $\sum_{j=1}^{i}$ KI[j].

```
proc  par_rsamp_case2  ( p,k; k,n,s,e ; SAMPLE );
var
   /* input */
   k,n,s,e : integer;
   /* output */
   SAMPLE : array [1..k] of 1..n;
   /* local */
   KI,SUMKI : array [1..k] of 1..k;
   GROUP : array [1..k] of 1..⌈n/k⌉

begin
I1: if ⌈n/k⌉.k <> n then breakpt := RAND[1,n]
I2: for i := p to p+k-1 pardo
       GROUP[i-p+1] := RAND[ 1,⌊n/k⌋ ]
    dopar;
I3: par_findki ( p,k ; n,k,GROUP ; GROUP,KI,T );
I4: par_find_parsum ( p,T ; KI,T ; SUMKI );
I5: for i := 1 to T pardo
       begin
          par_random_sample ( p+SUMKI[i-1],KI[i] ; KI[i],k,
                              SUMKI[i-1]+1, SUMKI[i] ; SAMPLE );
          for j := p+SUMKI[i-1] to p+KI[i]-1 pardo
             SAMPLE[j-p] := SAMPLE[j-p] + (GROUP[i]-1) * k
          dopar
       end
    dopar; → if ⌊n/k⌋·k <> n
I6: par_modify_sample (p,k ; k,n,breakpt,SAMPLE ; SAMPLE)
end;
```

The procedure **par_random_sample** presents the algorithm for selecting a random sample SAMPLE1[1:k] of size k out of n distinct items. It calls the procedures **par_rsamp_case1**, **par_rsamp_case2**, depending on whether $n \leq k^2$ or $n > k^2$.

```
proc  par_random_sample  ( p,k; k,n,s,e ; SAMPLE1 );
var
   /* input */
   k,n,s,e : integer;
   /* output */
   SAMPLE1 : array [1..k] of 1..N;
   /* local */
   SAMPLE : array [1..k] of 1..n;
```

```
begin
     if ⌈n/k⌉ ≤ k then
        par_rsamp_case1 (p,k ; k,n ; SAMPLE1);
     else
        begin
           par_rsamp_case2 (p,k; k,n,s,e; SAMPLE);
           for i := p to p+e-s pardo
              SAMPLE1 [i-p+s] := SAMPLE [i-p+1]
           dopar
        end
end;
```

LEMMA 2.5

The sample of k out of n items produced by algorithm **par_random_sample** is uniformly random.

proof:

If $\lceil n/k \rceil \leq k$ then by LEMMA 2.1 the sample is random. Therefore, the only case that needs to be considered is when $\lceil n/k \rceil > k$ at each stage of recursion.

If, at every stage, the number of items being selected from any group is a factor of the size of the group i.e. $n = \lceil n/k \rceil . k$, then by symmetry, any sample generated follows a uniform distribution. If at any stage the number of items to be selected is not a factor of the size of the group, a contiguous block of $(n - \lfloor n/k \rfloor . k)$ items is rejected. Since, one such block out of the possible n is chosen at random (assuming circularity among the items i.e. item 1 follows item n) , we are assured of a uniform distribution.

[]

LEMMA 2.6

The algorithm par_random_sample for selecting a sample of k items out of n at random runs in O(log k) time with

O(k) processors using O(k) memory.

proof:

Let T(k,n) be the time complexity for the random sampling problem. In case $\lceil n/k \rceil \leq k$ at any stage of recursion, the algorithm **par_rsamp_case1** is invoked and the process is completed in O(log k) time with O(k) processors. So the worst case is when $\lceil n/k \rceil > k$ (or $k^2 < n$) at all stages of recursion (except the last), in which case every time the procedure **par_rsamp_case2** is invoked. Therefore, the worst case value of the number of elements to be selected at (say) i^{th} stage is squareroot of the number of elements to be selected at $(i-1)^{th}$ stage. All steps of the procedure **par_rsamp_case2** except the recursive step can easily be implemented in O(log k) time. Thus,

$$T(k,n) := T(\text{sqrt}(k),k) + O(\log k)$$

$$\text{or, } T(k,n) := O(\log k)$$

Since at any stage of recursion the total number of items being selected does not exceed k, the processor complexity of the algorithm is O(k).

Also, the space complexity of the algorithm is given by:

$$S(k) := S(\text{sqrt}(k)) + O(k)$$

$$\text{or, } S(k) := O(k)$$

[]

2.3. ANOTHER APPROACH TO PARALLEL RANDOM SAMPLING

In this section, an algorithm **par_random_sample2** for selecting k items at random out of n distinct items is presented which uses an approach different from the divide and conquer approach of the algorithm **par_random_sample** of last section.

GOODMAN & HEDETNIEMI [GH77] have presented algorithm **select** with a time complexity of O(n) and space complexity O(n). The algorithm, first initializes an array ITEM[1:n] with the integers 1 through n and a pointer 'last' to the last element i.e. n. Next, the algorithm proceeds in k steps. At each step an item is selected at random from ITEM[1:last], the item is replaced by the element pointed to by 'last', and the 'last' pointer is decremented by 1. The algorithm has been modified by ERNVALL & NEVALAINEN [EN82] such that their algorithm has a time complexity of $O(k^2)$ and space complexity O(k). TEUHOLA & NEVALAINEN [TN82] made further improvements to the space complexity. Their algorithm uses a chain of substitutions to get the indices of the sample. The algorithm uses a linear search technique and hence $O(k^2)$ computations. GUPTA & BHATTACHARJEE [GB84] use sorting searching and interchange methods to give an algorithm with time complexity O(k.log k). The algorithm needs two passes of sort,search and interchange.

The algorithm **isel** which is due to GUPTA & BHATTACHARJEE manipulates a set A =

{(SELECT[i],LAST[i],STAGE[i])|i:=1,..,k} ,where initially

SELECT[i] - contains a random integer RAND[1,n-i+1].

LAST[i] - contains the index of the last item (n-i+1).

STAGE[i] - contains i.

The output of the algorithm is SELECT[1:k] which contains the random sample of k out of n items. It has been shown in [GB84] that the algorithm isel generates the same sequence of items as algorithm select.

```
proc isel ( k,n ; SELECT );
var
   /* input */
   k,n : integer;
   /* output */
   SELECT : array [1..k] of 1..n;
   /* local */
   LAST,STAGE : array [1..k] of 1..n;
   i,j : integer
begin
I1 : for i := 1 to k do
        begin
           SELECT[i] := RAND [1,n-i+1];
           LAST[i] := n-i+1;
           STAGE[i] := i
        end;
I2 : /* sort */
     Arrange the elements of set A such that for i < j
     either  SELECT[i] > SELECT[j]
         or  SELECT[i] = SELECT[j] and
             LAST[i] > LAST[j]
I3 : /* search and interchange in forward direction */
     for i := 1 to k-1 do
        if SELECT[i] = SELECT[i+1] then
           begin
              interchange (SELECT[i],LAST[i]);
              interchange (STAGE[i],STAGE[i+1])
           end;
I4 : /* sort */
     Arrange the elements of set A such that for i < j
     either SELECT[i] > SELECT[j]
         or SELECT[i] = SELECT[j] and
            LAST[i] > LAST[j]
```

```
I5 : /* search and interchange in backward direction */
     for i := 1 to k-1 do
        begin
           j := k-i+1;
           find l, 1 in [1,k-i], SELECT[j] = SELECT[l]
           if found then
              begin
                 interchange (SELECT[l],LAST[l]);
                 interchange (STAGE[j],STAGE[l])
              end
        end
I6 : Sort the elements of A such that for i < j
        STAGE[i] < STAGE[j]
end;
```

The algorithm **par_random_sample2** is based upon an interesting observation, which is stated in the following lemma, in GUPTA & BHATTACHARJEE's algorithm **isel**.

LEMMA 2.7

After steps I1-I3 of algorithm **isel**, the sample SELECT[i], $1 \leq i \leq k$, is such that each item occurs at most twice in the sample.

proof:

In step I1, SELECT[i] is set to RAND[1,n-i+1] and LAST[i] to n-i+1. Thus each element of LAST[1:k] is distinct from the other. If any item has more than one occurence in SELECT, after the sorting step all the occurences are brought together. In step I3, each repeated occurence of any item is replaced by the corresponding element of LAST. There is at most one occurence of any SELECT[i] in LAST. Therefore, after the interchange of step I3, each element of array SELECT may be found to repeat at most once.

[]

The above lemma suggests an algorithm which can solve the random sampling problem in two passes of steps I1-I3. In the first pass at least k/2 (and at most k) items are selected for the sample. Out of the remaining items, selecting another sample of size which is twice the number of items required, results in the remaining items. Combining the results of the two passes we get a random sample of size k out of n items. It is assumed that the sample generated by the first pass is withdrawn before starting the second pass (In the actual implementation of the second pass, one may first generate a sample consisting of the *ranks* of the items of the second sample in the initial set of n items from which the first sample has been withdrawn. Later, the second sample can be generated from among the gaps between successive items of the first sample). This ensures that none of the items selected in the first pass are selected in the second. The second pass necessitates that (at most) k items be remaining after the first pass (which selects at least k/2 items). This imposes a limit to n i.e. $n > 3k/2$ to start with. However this is not a serious constraint because in case $n \leq 3k/2$, the problem of selecting k out of n items can be converted to the problem of rejecting n-k items out of n.

Based upon the above discussion a parallel algorithm can be designed. The algorithm involves the following major steps:

1. Generation of the first sample consisting of at

least k/2 items in pass 1.

2. Generation of the ranks of the items to be selected for the second sample under the assumption that the first sample has already been withdrawn.

3. Generation of actual items present in the second sample from the ranks generated in step 2.

4. In case $n \leq 3k/2$ at the start, finding the complement set of the sample generated (since in this case the sample generated has to be rejected).

A parallel implementation of the above algorithm is presented next:

The main algorithm is presented in procedure **par_random_sample2.** This in turn calls the procedures **par_rsamp2_pass1, par_rsamp2_pass2, par_find_sample2, par_complement,** which correspond to the steps 1-4 mentioned above. The procedure **par_rsamp2_pass1** generates the sample of the first pass SAMP1[1:npass1] where 'npass1' is the number of distinct items selected by pass 1. It in turn calls the procedure **par_compress,** presented in chapter 1, to compresses the distinct elements of SAMP1[1:k] into SAMP1[1:npass1]. The second pass of the algorithm may produce more than the required number of items. This condition is taken care of by the procedure **par_find_endpt** inside **par_rsamp2_pass2.** The procedure **par_find_sample2** calls the procedure **par_find_actindx** which, for each element of the second sample, does a binary search over the gaps between the successive items of the sample generated by the

first pass to find the actual item corresponding to it.

```
proc  par_rsamp2_pass1  ( p,k ; k,n ; SAMP1,npass1 );

var
   /* input */
   k,n : integer;
   /* output */
   SAMP1 : array [1.. npass1] of 1..n;
   npass1 : integer;
   /* local */
   i,u : integer;
   SELECT,LAST : array [1..k] of 1..n
begin
I1: /* initialize */
    for i := p to p+k-1 pardo
       begin
          u := i-p+1;
          SELECT[u] := RAND [1,n-u+1];
          LAST[u] := n-u+1
       end
    pardo;
I2: /* sort */
    Arrange the elements of set
    A = {(SELECT[i],LAST[i]) | i := 1..k} such that
    for i < j either SELECT[i] > SELECT[j]
                  or SELECT[i] = SELECT[j] and
                     LAST[i] > LAST[j];
I3: for i := p to p+k-2 pardo
       begin
          u := i-p+1;
          if SELECT[u] = SELECT[u+1] then
             SELECT[u] := LAST[u];
          SAMP1[u] := SELECT[u]
       end
    dopar;
I4: par_compress ( p,n ; SAMP1,k,n ; SAMP1,npass1 )
end;
```

The procedure **par_rsamp2_pass2** requires that the sample SAMP2 generated by it should have exactly 'npass2' item indices. However, steps I1-I5 of the procedure may generate more than 'npass2' distinct items. So, the procedure **par_find_endpt** is run to find 'endpt' such that SAMP2[1:endpt] yields a sample of exactly 'npass2' distinct

items.

```
proc  par_find_endpt  (p,newn;
                       SELECT,STAGE,newn,npass2,n;endpt);
var
   /* input */
   SELECT : array [1..newn] of 1..n;
   STAGE : array [1..newn] of 1..newn;
   npass2,newn,n : integer;
   /* output */
   endpt : integer;
   /* local */
   i,u : integer;
   BIT : array [1..newn] of 0..1;
   FAR : array [1..newn] of 1..newn

begin
I1: Sort the elements of the  set A =
    {(SELECT[i],STAGE[i]) | i := 1,..,newn} such that
    for i < j either SELECT[i] > SELECT[j]
                  or SELECT[i] = SELECT[j] and
                     STAGE[i] < STAGE[j];
I2: for i := p+1 to p+newn-1 pardo
       begin
          u := i-p+1;
          if SELECT[u] = SELECT[u-1] then
             BIT[STAGE[u]] := 0
          else
             BIT[STAGE[u]] := 1
       end
    dopar;
    BIT[STAGE[1] := 1;
I3: for i := p to p+newn-2 pardo
       FAR[i-p+1] := i-p+2
    dopar;
    FAR[newn] := newn;
I4: for ⌈log newn⌉ steps do
       for i := p to p+newn-1 pardo
          begin
             u := i-p+1;
             BIT[FAR[u]] := BIT[u] + BIT[FAR[u]];
             FAR[u] := FAR[FAR[u]]
          end
       dopar;
I5: for i := p to p+newn-1 pardo
       if BIT[i-p+1] = npass2 then endpt := i-p+1
    dopar
end;
```

The procedure **par_rsamp2_pass2** gives the algorith for the second pass wherein SAMP2[1:npass2] is selected.

```
proc  par_rsamp2_pass2  ( p,np ; npass2,n ; SAMP2 );
var
   /* input */
   npass2,n : integer;
   /* output */
   SAMP2 : array [1..2*npass2] of 1..n;
   /* local */
   LAST,SELECT : array [1..2*npass2] of 1..n;
   STAGE : array [1..2*npass2] of 1..2*npass2;
   i,u,newn,tnum,endpt : integer
begin
I1: newn := 2 * npass2;
I2: for i := p to p+newn-1 pardo
       begin
          u := i-p+1;
          SELECT[u] := RAND [1,n-u+1];
          LAST[u] := n-u+1
       end
    dopar;
I3: Arrange elements of the set A =
    {(SELECT[i],LAST[i]) | i := 1,..,k} such that
    for i < j either SELECT[i] > SELECT[j]
                  or SELECT[i] = SELECT[j] and
                     LAST[i] > LAST[j];
I4: for i := p to p+newn-2 pardo
       begin
          u := i-p+1;
          if SELECT[u] = SELECT[u+1] then
             SELECT[u] := LAST[u]
       end
    dopar;
I5: for i := p to p+newn-1 pardo
       begin
          u := i-p+1;
          SAMP2[u] := SELECT[u];
          STAGE[u] := u
       end
    dopar;
I6: par_find_endpt (p,newn; SELECT,STAGE,newn,npass2,n; er
I7: par_compress (p,endpt; SAMP2,endpt,n; SAMP2,tnum);
end;
```

The sample SAMP2 generated by **par_rsamp2_pass2** i step I6 of the main procedure (**par_random_sample2**) doe not contain the actual indices of items but their ranks i

the items remaining after the first sample SAMP1 has been removed. The procedure **par_find_sample2** finds the actual item indices for the items selected in the second pass.

```
proc  par_find_sample2  (p,npass1 ;
                    npass1,npass2,n,k,SAMP1,SAMP2;SAMP2);
var
   /* input */
   npass1,npass2,n : integer;
   SAMP1,SAMP2 : array [Ø..k+1] of 1..n;
   /* output */
   SAMP2 : array [1..npass2] of 1..n;
   /* local */
   i,u : integer;
   GAP : array [Ø..npass1+1] of 1..n-npass1;
   FAR : array [1..npass1+1] of 1..npass1+1
begin
I1: for i := p+1 to p+npass1-1 pardo
       begin
          u := i-p+1;
          GAP[u] := SAMP1[u] - SAMP1[u-1]
       end;
    GAP[Ø] := Ø;
    GAP[1] := SAMP1[1] - 1;
    GAP[npass1+1] := N+npass1-SAMP1[npass1];
I2: for i := p to p+npass1-1 pardo
       FAR[i-p+1] := i-p+2
    dopar;
    FAR[npass1+1] := npass1+1;
I3: for ⌈log(npass1+1)⌉ steps do
       for i := p to p+npass1 pardo
          begin
             u := i-p+1;
             GAP[FAR[u]] := GAP[FAR[u]] + GAP[u];
             FAR[u] := FAR[FAR[u]]
          end
       dopar;
I4: par_find_actindx (p,npass2 ;
end;         GAP,SAMP2,SAMP1,npass2,npass1,n ; SAMP2);
```

The procedure par_find_actindx, for each item index in SAMP2, does a binary search over the gaps present between successive elements in the sorted SAMP1 to find the actual index corresponding to it.

```
proc  par_find_actindx  (p,npass2;GAP,SAMP2,SAMP1,
                                  npass2,npass1,n;SAMP2);
var
   /* input */
   GAP : array [Ø..k] of 1..n;
   SAMP1 : array [1..k+1] of 1..n;
   SAMP2 : array [1..npass2] of 1..n;
   npass2,npass1,n : integer;
   /* output */
   SAMP2 : array [1..k] of 1..n;
   /* local */
   LO,UP,MI : array [1..npass2] of 1..npass1+1
begin
   for i := p to p+npass2-1 pardo
      begin
         u := i-p+1;
         LO[u] := 1;
         UP[u] := npass1+1;
         while LO[u] < UP[u] do
            begin
               MI[u] := ⌊(LO[u] + Up[u])/2⌋;
               if ( GAP[MI[u]] > SAMP2[u] ) and
                  ( GAP[MI[u]-1] < SAMP2[u] ) then
                  begin
                     SAMP2[u] := SAMP1[MI[u]] -
                          (GAP[MI[u]]-SAMP2[u]) - 1;
                     return
                  end
               else
                  if SAMP2[u] > GAP[MI[u]] then
                     LO[u] := MI[u] + 1
                  else
                     UP[u] := MI[u] - 1
            end
      end
   dopar
end;
```

In case $n \leq 3.k/2$, the problem of selection is converted to the problem of rejecting n-k elements out of n. In such a situation, at the end of the algorithm, the procedure **par_find_complement** finds the complement of the sample generated with respect to the total population. It is to be noted that whenever this procedure is run O(n) = O(k).

```
proc  par_find_complement  (p,n; SAMPLE,k,n; SAMPLE);
var
   /* input */
   SAMPLE : array [1..n-k] of 1..n;
   k,n : integer;
   /* output */
   SAMPLE : array [1..n-k] of 1..n;
   /* local */
   NSAMPLE : array [1..n] of 1..n;
   BIT : array [1..n] of Ø..1;
   i,u : integer

begin
I1: for i := p to p+n-1 pardo
       begin
          u := i-p+1;
          BIT[u] := 1;
          NSAMPLE[u] := u
       end
    dopar;
I2: for i := p to p+k-1 pardo
       BIT[SAMPLE[i-p+1]] := Ø
    dopar;
I3: Sort the set
    B := {(BIT[i],NSAMPLE[i]) | i := 1,..,n} such that
    for i < j ,  BIT[i] ≥ BIT[j];
I4: for i := p to p+n-k-1 pardo
       begin
          u := i-p+1;
          SAMPLE[u] := NSAMPLE[u]
       end
    dopar
end;
```

The procedure **par_random_sample2** presents the algorithm for selecting a random sample SAMPLE[1:k] of size k out of n distinct items.

```
proc  par_random_sample2  (p,k ; k,n ; SAMPLE );
var
   /* input */
   k,n : integer;
   /* output */
   SAMPLE : array [1..k] of 1..n;
   /* local */
   i,npass1,npass2,flag : integer;
   SAMP1,SAMP2 : array [1..k] of 1..n
```

```
begin
I1: if n ≤ (3*k)/2 then
       begin
          flag := 1;
          k := n-k
       end
    else
       flag := Ø;
I2: par_rsamp2_pass1 (p,k ; k,n ; SAMP1,npass1);
I3: n := n-npass1;
I4: npass2 := k-npass1;
I5: if npass2 > Ø then
       begin
I6:       par_rsamp2_pass2 (p,2*npass2 ; npass2,n ;SAMP2);
I7:       par_find_sample2 (p,npass1;
                    npass1,npass2,n,SAMP1,SAMP2;SAMP2)
       end;
I8: for i := p to p+npass1-1 pardo
       SAMPLE[i-p+1] := SAMP1[i-p+1]
    dopar;
I9: for i := p to p+npass2-1 pardo
       SAMPLE[npass1+i-p+1] := SAMP2[i-p+1]
    dopar
I1Ø: if flag = 1 then
        par_find_complement(p,n; SAMPLE,k,n; SAMPLE)
end;
```

LEMMA 2.8

All the items selected for the sample by the procedure **par_random_sample2** are distinct and lie between 1 and n.

proof:

Pass 1 of the algorithm (**par_rsamp2_pass1**) is essentially the same as steps I1-I3 of algorithm **isel**, and thus by LEMMA 2.1 it generates a sample (SAMP1) of size 'npass1' ($k/2 \leq npass1 \leq k$) such that each index selected is distinct. The remaining (k-npass1) items are selected in Pass 2 (**par_rsamp2_pass2**) by generating 2 * (k-npass1) random integers in the range [1,n-npass1]. In case more than k-npass1 distinct integers are generated, we consider only the first 'endpt' processors (i.e. those numbered 1..endpt)

such that the random integers generated by these form a sample of exactly (k-npass1) distinct item indices. Thus, after the two passes are over we have two samples of sizes npass1 and (k-npass1) respectively such that each contains distinct item indices. Next, the two samples (stored in sorted order in SAMP1 and SAMP2 respectively) are combined as follows :

The assumption is that SAMP1 is withdrawn before SAMP2 i.e. SAMP2[i] gives the position of the i^{th} item of the second sample after the first sample has been withdrawn from the pool of n items. The actual item index corresponding to each SAMP2[i] is found as follows: Gaps between successive elements of SAMP1 are found. By Binary search the actual gap in which each SAMP2[i] lies (and hence the actual index corresponding to SAMP2[i]) is found. Since, the indices thus selected can lie only in the gaps of SAMP1, they are distinct from the sample items in SAMP1. Thus in the combined sample, each index is different from the other.

Also, the maximum value of any SAMP2[i] is n-npass1. Since the size of SAMP1 is 'npass1', the maximum index corresponding to any SAMP2[i] is n-npass1+npass1 := n. The minimum value of any SAMP2[i] is 1 and thus the minimum item index corresponding to it is 1+Ø := 1.

[]

LEMMA 2.9

The algorithm par_random_sample2 requires O(log k) time using O(k) processors.

proof:

Each of the steps of the procedures **par_random_sample2, par_rsamp2_pass1, par_rsamp2_pass2, par_compress, par_find_endpt, par_find_actindx par_find_complement, par_find_sample2** may be executed in at most O(log k) time with O(k) processors. For parallel sorting may use the algorithm in [CO86], which run in O(log k) time with O(k) processors.

[]

2.4. RANDOM PERMUTATION IN PARALLEL

With the help of the random sampling algorithm one can easily design an algorithm for getting a Random Permutation of n items. The problem is defined as follows :

Given a sequence of n items A = (a_1 a_2 . . a_n), generate a permutation of the n items such that the position of occurence of each item is random. Without loss of generality one may consider the n items to be the integers 1..n.

One may generate a random permutation by generating the k^{th} permutation in a lexicographic ordering, such that k is chosen at random. However, this process is inefficient since the best known sequential algorithm takes O(nlog n) time [GB83]. One may generate a random permutation in O(n) time on a serial computer by a sequence of n interchanges such that the i^{th} interchange swaps A[i] and A[j] ,where j is some index selected at random to the right of i [NW75].

```
proc  rand_perm  (A,n;A);

var
   /* input */
   n : integer;
   A : array [1..n] of 1..n;
   /* output */
   A : array [1..n] of 1..n;
   /* local */
   i,r,temp : integer;
begin
I1: for i := 1 to n do
       begin
          r := RAND [Ø,1]; /* r is a random variate
                              in the the open interval
                              (Ø,1) */
          l := i + ⌊r.(n+1-i)⌋;
          temp := A[l];
          A[l] := A[i];
          A[i] := temp
       end;
```

The above approach however is not suitable for parallel computation because we may get a chain of interchanges which have to be done sequentially. But given the results of the last two sections, one may generate a random permutation in polylog time using procedure **par_random_perm** (A,1,n). The algorithm proceeds by routing n/2 randomly selected items into the first half of the array A and putting the remaining items into the second half of A. The algorithm is then called recursively for both the halves of A simultaneously. In case n is odd, one item selected at random is made to be the last element.

```
proc  par_random_perm  (p,j-i+1; A,i,j; A);
var
   /* input */
   i,j : integer;
   A : array [1..n] of 1..n;
   /* output */
   A : array [1..n] of 1..n;
   /* local */
```

```
   A1,A2 : array [1..n] of 1..n;
begin
I1: if (j-i+1) is odd then
       begin
          term := RAND[i,j];
          temp := A[term];
          A[term] := A[j];
          A[j] := temp;
          j := j-1
       end
    else
       begin
I2:       k := ⌊(j-i+1)/2.0⌋;
I3:       par_random_sample (p,k; k,j-i+1,i,j; A1);
I4:       A2 := A-A1; /* A2 is an array containing those
                         elements of A which are not in A1 */
I5:       for m := p to p+k-i pardo
             begin
                u := m-p;
                A[i+u] := A1[u];
                A[k+1+u] := A2[u]
             end;
I6:       parbegin
             par_random_perm(p,k ; A,i,k ; A);
             par_random_perm(p,j-k; A,k+1,j ; A)
          parend
       end
end;
```

LEMMA 2.10

The procedure **par_random_perm** takes $O((\log n)^2)$ time to compute a random permutation of (1,..n) using O(n) processors.

proof:

Steps I2,I3 take O(log n) time using O(n) processors, while step I1 takes constant time. Step I4 constitutes the recursive step. Thus, the time complexity of the algorithm is given by:

$$T(n) := T(n/2) + O(\log n)$$

or, $T(n) := O((\log n)^2)$.

2.5. CONCLUSION

In this chapter two parallel algorithms for generating a Random Sample of size k out of a population of size n are presented. Both the algorithms, algorithm **par_random_sample** and **par_random_sample2** proceed in O(log k) time when O(k) processors are available. The cost (product of time and processor complexities) is same as the best known sequential algorithm using O(k) space which runs in O(k.log k) time.

The random permutation of k objects can be generated in $O((\log k)^2)$ time with O(k) processors using procedure **par_random_perm**. However, if a random permutation of k out of n objects is required, one can first generate a random sample of size k out of n.

CHAPTER 3
GENERATION OF RANDOM UNLABELED ROOTED TREE & RANDOM BINARY TREE

3.1. INTRODUCTION

The problem of generation of trees, as combinatorial objects, has received substantial attention in literature. Trees may be rooted or unrooted, labeled or unlabeled. Rooted trees may be ordered or unordered. There are other restricted classes of trees such as binary, k-ary etc. Considerable work has been done in generating trees under various coding schemes [R78],[ZR79],[Z8Ø]. Since there may be an exponential number of trees belonging to any particular class, a natural question that arises is how we can generate a random tree belonging to any such class. If the exact number of trees in any class are known then selection of a random tree can be done via a random number generator which generates a random number between 1 and |T(n)|, where |T(n)| is the total number of trees in the given class. The tree corresponding to the number can then be found in the given class of trees. This process of generating a tree corresponding to a rank in some ordering of trees is termed as *unranking*. But, the unranking procedures are usually inefficient. So novel procedures need to be developed which generate a random tree of a particular class. In this chapter we deal with two such classes of trees, the unla-

beled rooted trees and binary trees.

A parallel algorithm par_ran_tree for the problem of generating a random unlabeled rooted tree having n nodes with number of levels (see section 3.2) constrained by some k, $1 \leq k \leq n$, and which runs in O(log k) time with O(n) processors is presented in section 3.2. In section 3.3, two optimal sequential algorithms ran_bintree1 and ran_bintree2 are presented for the problem of generating a random binary tree having n nodes. While ran_bintree1 generates a binary tree with an unconstrained number of levels, ran_bintree2 generates a tree with number of levels bounded by some k, $\lceil \log n \rceil \leq k \leq n$.

3.2 PARALLEL ALGORITHM FOR GENERATION OF RANDOM UNLABELED ROOTED TREE

In this section, a parallel algorithm is presented for generating a random unlabeled rooted tree having n nodes and k levels. The *level of a node* is defined to be equal to one more than the length of the path from the node to the root of the tree. The *number of levels in a tree* may be defined to be the level of the node having the largest level in the tree. The algorithm can be extended to work for the case of random labeled tree by taking a random permutation of the node numbers. The term *unlabeled rooted tree* is used because the algorithm does not distinguish between trees which are isomorphic but have a different labeling for the n nodes.

3.2.1 RESULTS CONCERNING SEQUENTIAL GENERATION

A sequential algorithm due to Nijenhuis & Wilf [NW75] exists for the above problem. The algorithm runs as follows:

step1:Select an integer m , $1 \leq m \leq n$, at random.

step2:Select a divisor d of m, at random.

step3:Generate a random unlabeled tree T' having n-m nodes.

step4:Generate a random unlabeled tree T'' having m nodes.

step5:Make j := m/d copies of T''

step6:Join the root of T' to the roots of each of the copies of T''.

Steps 3 & 4 constitute the recursive calls to the procedure. In the worst case, the depth of recursion is O(n) (when m = n-1 at each stage of recursion). Thus, at least in its present form, this algorithm cannot be efficiently parallelised in a straightforward manner. Also, if one wants a tree with k levels one may have to run this algorithm more than once. However, the parallel algorithm of section 3.2.2 directly generates a tree with k levels.

Similarly, a sequential algorithm exists for generating a random labeled tree which is a direct implementation of Prufer's proof of Cayley's theorem [NW75]. However, it seems that this also cannot be efficiently parallelised.

3.2.2 ALGORITHM par_ran_tree

The algorithm presented here makes use of the parallel random sampling algorithm of the previous chapter to partition the n nodes into k parts, at random. The first part of the partition is set equal to 1 since, at the first level there should be only one node which is the root of the tree. Besides, each part must be greater than Ø, otherwise the number of levels in the tree becomes less than k. So, initially one node is assigned to each of the remaining k-1 parts. For partitioning the remaining n-k nodes into k-1 parts the algorithm proceeds as follows: Let us assume that we have n-2 in unary form (i.e. as a sequence of n-2 ones). Out of these we select k-2 1's at random (using par_random_sample2). These act as the boundaries for the k-1 parts into which n-k has to be divided. The number of 1's between the i^{th} boundary and the $(i+1)^{th}$ boundary is the size of the $(i+2)^{th}$ part. It is assumed that the second and the k^{th} parts are delimited on their left and right respectively by boundaries situated at positions Ø and n-1 respectively. Thus we conclude as follows:

LEMMA 3.1

The n nodes can be partitioned into k parts (each part greater than Ø and the first part equal to 1), such that the i^{th} part corresponds to the number of nodes at the i^{th} level of the tree.

[]

LEMMA 3.1 is implemented by the procedure **par_partition**. This procedure in turn calls the procedure **par_random_sample2** presented in the last chapter for finding a random sample.

```
proc par_partition  ( p,n ; n,k ; PART );
var
  /* input */
  n,k : integer;
  /* output */
  PART : array [1..k] of 1..n-k+1;
  /* local */
  BOUND : array [1..k-2] of 1..n-2;
  u,i:integer

begin
I1: par_random_sample2 ( p,k-2 ; k-2,n-2 ; BOUND );
I2: for i := p to p+k-1 pardo
      begin
         u := i-p+1;
I3:       if u = 1 then
            begin
               PART[u] := 1;
               PARTSUM[u] := 1
            end
          else
            if u = 2 then
               b egin
                  PART[u] := BOUND[1];
                  PARTSUM[u] := BOUND[1] +1
               end
            else
              if u = k then
                 begin
                    PART[u] := n+k-3 - BOUND[k-2] + 1;
                    PARTSUM[u] := n;
                 else
                    begin
                       PART[u] := BOUND[u-1] - BOUND[u-2] + 1;
                       PARTSUM[u] := BOUND[u-1] + 1
                    end
      end
   dopar
end;
```

After obtaining the number of nodes at each level i, every node at i^{th} level randomly selects a node in the $(i-1)^{th}$ level as its parent.

LEMMA 3.2

If, for each node in the i^{th} part of the partition generated by procedure **partition**, a node in the $(i-1)^{th}$ level is selected at random as its PARENT, then an unlabeled rooted tree is generated.

proof:

The PARENT relation, along with the specification of the root node completely defines the rooted unlabeled tree.

[]

The procedure **par_ran_tree** implements LEMMA 3.2. i.e. corresponding to each node at level i, $1 < i \leq k$, it finds and fixes, at random, another node at level i-1 as the parent of the former node.

```
proc par_ran_tree  ( p,n ; n,k ; PARENT,root );
var
  /* input */
  n,k : integer;
  /* output */
  root : integer;
  PARENT : array [1..n] of 1..n;
  /* local */
  PART : array [1..k] of n-k+1;
  PARTSUM : array [1..k] of 1..n;
  i,j,u,v : integer
begin
I1: par_partition ( p,n ; n,k ; PART,PARTSUM );
    PARTSUM[Ø] := Ø;
I2: for i := p+1 to p+k-1 pardo
      begin
        u := i-p+1;
        for j := p + PARTSUM[u-1] to
                    p + PARTSUM=PART[u]-1 pardo
          begin
            v := j-p+1;
            PARENT[v] := RAND[ PARTSUM[u-2]+1,
                               PARTSUM[u-1] ]
          end
        dopar
      end
```

```
    dopar;
I3: root := 1;
    PARENT[root] := root
end;
```

LEMMA 3.3

The tree generated by the procedure **par_ran_tree** is random.

proof:

Since the initial division of n nodes among k levels and the subsequent selection of PARENT relation for each node is done at random, the tree generated is random.

[]

LEMMA 3.4

Procedure **par_ran_tree** generates a random unlabeled rooted tree having n nodes and k levels in O(log k) time with O(n) processors.

proof:

The procedure **par_partition** takes O(log k) time with O(n) processors for the procedure **par_random_sample2**. All other steps can be completed in constant time with O(n) processors.

[]

The above parallel algorithm suggests a new sequential algorithm which is optimal in time and space complexities both of which are O(n). The algorithm may use the sequential algorithm **select** [GH77] presented in chapter 2 for divid-

ing n into k parts. The selection of a parent for each node can easily be completed in O(n) time.

COROLLARY

A Random labeled rooted tree with n nodes and k levels can be generated in $O((\log n)^2)$ time with O(n) processors.

[]

3.3. RANDOM BINARY TREE GENERATION

In this section, two sequential algorithms **ran_bintree1** and **ran_bintree2** for generating a binary tree with n nodes are presented. The algorithm **ran_bintree1** generates a binary tree with an unconstrained number of levels, while the algorithm **ran_bintree2** generates a tree whose number of levels is constrained by some k, where $\lceil \log n \rceil \leq k \leq n$. Both the algorithms are optimal in their time and space complexities, which are O(n).

3.3.1 ALGORITHM ran_bintree1

This algorithm builds up the binary tree level by level till all the nodes are exhausted. For each level, the algorithm uses procedure **select** to select, at random, the nodes which have left sons or right sons at the next level.

```
proc select ( k,n ; SAMPLE );
var
  /* input */
  k,n : integer;
  /* output */
  SAMPLE : array [1..k] of 1..n;
  /* local */
  ITEM : array [1..n] of 1..n;
```

```
  i,s,last :integer
begin
  for i := 1 to n do
     ITEM[i] := i;
  last := n;
  for i := 1 to k do
     begin
        s := RAND [ 1,last ];
        SAMPLE[i] := ITEM[s];
        ITEM[s] := ITEM[last];
        last := last-1
     end
end;
```

LEMMA 3.5

The procedure select generates a random sample of k out of n items in O(n) time using O(n) space.

proof:

Refer to [GH77].

[]

Let the number of nodes at the i^{th} level is given by NUM_NODE[i]. Assume that we have built the binary tree upto the i^{th} level and we want to build the $(i+1)^{th}$ level. The choice of NUM_NODE[i+1] depends upon the following constraints:

1. NUM_NODE[i+1] belongs to the interval

   $[1, n - \sum_{j=1}^{i} \text{NUM\_NODE}[j]]$

2. $1 \leq \text{NUM\_NODE}[i+1] \leq 2*\text{NUM\_NODE}[i]$

After selecting NUM_NODE[i+1] at random satisfying the above constraints, the LSON and RSON relations for the i^{th} level have to be decided. The NUM_NODE[i+1] nodes are

divided into two parts L[i+1] and R[i+1], the number o nodes which form the left sons and those which form th right sons of the nodes at the i^{th} level respectively. Th only constraint which has to be satisfied is that neithe L[i+1] nor R[i+1] should be greater than NUM_NODE[i]. Having selected L[i+1] and R[i+1], the random sampling algorithm **select** is used to select L[i+1] and R[i+1] nodes respectively (stored in LSELECT and RSELECT respectively) from NUM_NODE[i]. If some node is selected as one of the L[i+1], it has a left son in the tree. The case of nodes having right sons is similar. The process is continued till the number of nodes remaining for the next level is reduced to zero. The process ensures that every node except the root is either the left son or the right son of some other node. Besides, each node can have at most two children.

The procedure **ran_bintree1** described below is just a straightforward implementation of the technique described above for generation of a random binary tree.

```
proc ran_bintree1  ( n ; root, LSON, RSON );
var
   /* input */
   n: integer;
   /* output */
   root : integer;
   LSON,RSON : array [1..n] of Ø..n-1;
   /* local */
   level,tnode,tempn,i : integer;
   L,R,LSAMPLE,RSAMPLE,NUM_NODE : array [1..n] of 1..n;

begin
I1: for i := 1 to n do
       begin
          LSON[i] := Ø;
          RSON[i] := Ø
       end;
```

```
    level := 1; tnode := 1; root := 1; tempn := n-1;
    NUM_NODE[Ø] := Ø; NUM_NODE[1] := 1;
I2: while tempn > Ø do
       begin
          level := level + 1;
          NUM_NODE[level] := RAND [ 1,
                     min (2 * NUM_NODE[level-1], tempn)];
          L[level] := RAND [1,min (NUM_NODE[level],
                                   NUM_NODE[level-1])];
          R[level] := NUM_NODE[level] - L[level];
I3:       select (L(level), NUM_NODE[level-1], LSAMPLE);
I4:       select (R(level), NUM_NODE[level-1], RSAMPLE);
I5:       for i := 1 to L[level] do
             LSON[NUM_NODE[level-2]+LSAMPLE[i]] :=
                                              tnode + i;
I6:       for i := 1 to R[level] do
             RSON[NUM_NODE[level-2]+RSAMPLE[i]] :=
                                   tnode + L[level] + i;
          tnode := tnode + NUM_NODE[level];
          tempn := tempn - NUM_NODE[level]
       end
end;
```

LEMMA 3.6

The binary tree generated by algorithm **ran_bintree1** is random.

proof:

The number of levels in the tree, the number of nodes at any level, and the parent of any node in the tree are all selected at random. The decision whether any node is to be the left son or the right son is also taken at random.

[]

LEMMA 3.7

The procedure **ran_bintree1** is optimal and takes O(n) time using O(n) space.

proof:

If 'tlevel' is the number of levels in the binary tree

generated, the while loop of step I2 is executed 'tlevel' times. Steps I3 and I4 take O(NUM_NODE[level-1]) time. Taking the summation over all levels of the tree, the total time spent executing these steps is O(n). Similarly, the time spent in steps I5 and I6 is O(n). All other steps take constant time. Thus the overall complexity of the algorithm is O(n), which is optimal since it takes at least O(n) time to output the n nodes of the binary tree. The space used is in the form of arrays LSON[1:n] and RSON[1:n], and the temporary stores LSAMPLE and RSAMPLE. Since at any time |LSAMPLE| + |RSAMPLE| does not exceed O(n), the space complexity of the algorithm is O(n), which again is optimal.

[]

3.3.2 ALGORITHM ran_bintree2

The algorithm presented in this section generates a random binary tree whose number of levels does not exceed some k, $\lceil \log n \rceil \leq k \leq n$. However, if k is chosen at random between $\lceil \log n \rceil$ and n, a tree with random number of levels can be generated.

The algorithm proceeds in O(n) steps. At the i^{th} step, the i^{th} node is inserted into the binary tree built so far, such that the position where the node is inserted is chosen at random. The algorithm is similar to the algorithm select in that it uses O(n) extra space to choose the position where the new node is to be inserted at each step. The algorithm also needs to maintain the level of each node

inserted, so as to meet the constraint on the number of levels of the tree. Two arrays POS[1:n+1] and LRBIT[1:n+1] are maintained, which together form the set A = {(POS[i],LRBIT[i])|i:=1,..,last}. Each element of the set is an ordered pair which signifies an empty position in the binary tree built so far. POS[i] stores the node number and LRBIT[i] shows whether the left son position (LRBIT[i] = Ø) or the right son position (LRBIT[i] = 1) is of POS[i] is vacant. If both the left son and right son positions of any node are vacant there are two entries corresponding to the node in the set. Whenever a new node 'node' has to be inserted into the tree, one of the positions is chosen at random from the set A. The 'last' entry of the set is made to replace the chosen entry. The level of 'node' is made equal to one more than that of its parent. If the level has reached the predefined maximum, no new entries are made into the set A. However, when this is not the case, two elements (node, Ø) and (node,1) are added to A.

LEMMA 3.8

The procedure described above and implemented in proc **ran_bintree2** generates a binary tree with n nodes whose number of levels is bounded by some k, $\lceil \log n \rceil \leq k \leq n$.

proof:

Every node inserted is either the left son or the right son of some other node. Also corresponding to any node there

may be at most two entries in the set A, one corresponding to the left son position and the other to the right son position. Therefore each node may have at most one left son and one right son. Thus the structure generated is a binary tree. If any node is inserted at the k^{th} level, the left and right son positions are not added to the set A of vacant positions and hence no node lies at the $k+1^{th}$ level.

[]

```
proc ran_bintree2  ( n ; root,LSON,RSON );
var
   /* input */
   n : integer;
   /* output */
   root : integer;
   LSON,RSON : array [1..n] of 1..n-1;
   /* local */
   LRBIT : array [1..n+1] of Ø..1;
   POS : array [1..n+1] of 1..n;
   LEVEL : array [1..n] of 1..n;
   i,sel,last : integer

begin
   for i := 1 to n do
      begin
         LSON[i] := Ø;
         RSON[i] := Ø
      end;
   LEVEL[1] := 1;
   root := 1;
   POS[1] := POS[2] := 1;
   LRBIT[1] := Ø;
   LRBIT[2] := 1;
   last := 2;
   for i := 2 to n do
      begin
         sel := RAND[1,last];
         if LRBIT[sel] = Ø then
            LSON[POS[sel]] := i
         else
            RSON[POS[sel]] := i;
         POS[sel] := POS[last];
         LRBIT[sel] := LRBIT[last];
         last := last-1;
         LEVEL[i] := LEVEL[POS[sel]] + 1;
         if LEVEL[i] < k then
```

```
            begin
               POS[last+1] := POS[last+2] := i;
               LRBIT[last+1] := 0;
               LRBIT[last+2] := 1;
               last := last + 2
            end
      end
end;
```

LEMMA 3.9

The binary tree generated by the procedure **ran_bintree2** is random.

proof:

Whenever a new node is added, its position is selected at random from the set of vacant positions which are all equally likely and hence the binary tree generated is random.

[]

LEMMA 3.10

The procedure **ran_bintree2** runs in optimal O(n) time using O(n) space.

proof:

Each insertion of a node into the binary tree generated thus far takes O(1) time for a total of O(n) time over all insertions. This is optimal as it takes O(n) time to output the tree. Since a binary tree having n internal nodes has n+1 leaves, the size of the arrays LRBIT and POS is at most n+1. Also, the arrays LSON and RSON take O(n) space. Thus the space complexity is also optimal.

[]

3.4. CONCLUSION

In section 2, a parallel algorithm was presented for generating a random unlabeled rooted tree having n nodes and k levels (1<k<n) in O(log k) time with O(n) processors. The result was generalized to the case of random labeled rooted tree, which could be computed in $O((\log n)^2)$ time using O(n) processors. In section 2, two optimal sequential algorithms for generating a random binary tree with n nodes was presented. Out of the two, algorithm **ran_bintree2** was able to generate a binary tree with at most k levels. Both the algorithms ran in O(n) time and used O(n) space. The idea used in the case of unlabeled rooted trees cannot be generalised to binary trees since they have the constraint that the number of nodes at the i^{th} level are at most twice the number of nodes at $(i-1)^{th}$ level. Therefore the $(i+1)^{th}$ level cannot be generated before the i^{th} and this makes the process sequential.

CHAPTER 4
GENERATION OF BINARY TREES

4.1. INTRODUCTION

Binary tree, as a data structure, appears extensively in various applications in computer science. Thus, the problem of generation of binary trees has motivated considerable research activity [PR80],[LLW86].

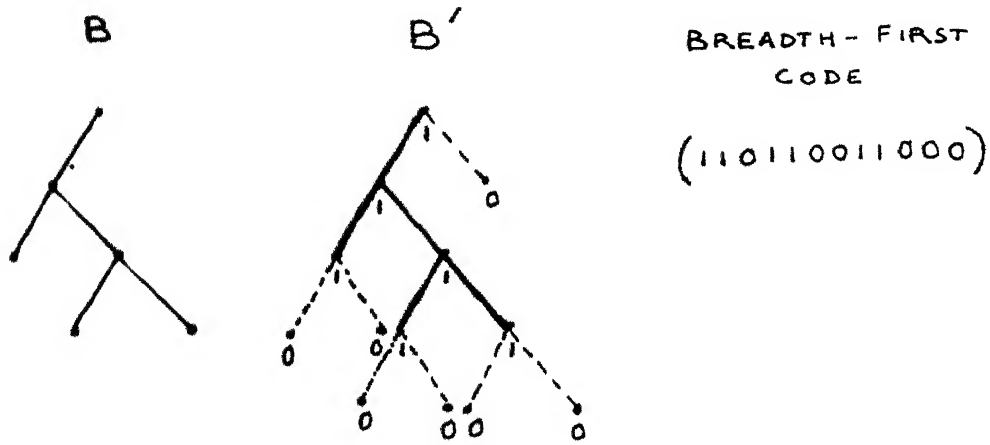
In this chapter a coding scheme has been described for binary trees and two algorithms for generation of binary trees in lexicographic order on a parallel machine are proposed. The first algorithm generates the trees one after the other in lexicographic order. The second algorithm generates all trees simultaneously. Both the algorithms achieve optimal speedup ratios.

In section 4.2 the theoretical background for the problem is developed while section 4.3 is concerned with presentation of the algorithms for generation of binary trees.

4.2.DEFINITIONS AND THEORETICAL BACKGROUND

Let us consider a binary tree B with n nodes. Consider another binary tree B' which is generated from B by adding the n+1 leaf nodes to it. B' may be defined to be the 2-ary form of B. The nodes of B in B' are the internal nodes. If we label the internal nodes of B' by 1 and the leaf nodes by Ø and read out all the labels in breadth-first order, a

code of size 2n + 1 is generated for the tree B. Neglecting the label corresponding to the rightmost leaf node, which will always be Ø, we get a code of size 2n. An example is shown in the figure.



The **lexicographic** order of generation of binary trees is the dictionary order i.e. the one in which, given the codes for any two binary trees, the numerically smaller tree is generated before the larger one. The **ranking** and **unranking** problems for binary trees are defined as follows:

Ranking - Given the code for a binary tree, find its rank in the lexicographic order of generation.

Unranking - Given a rank k, find the code of the binary tree corresponding to it in the lexicographic order of generation.

The terms 'code of a binary tree' and 'binary tree' have been used interchangeably in this chapter.

LEMMA 4.1

The code for the binary trees produced as above has the following properties:

1. Number of 1's = Number of Ø's = n.
2. In any prefix of the code, the number of 1's is greater than or equal to the number of Ø's.

proof:

The first property is obvious since there are n internal nodes and n+1 leaf nodes of the binary tree and we have neglected the last leaf node. If any prefix of the code is taken it corresponds to a binary tree which has not been fully converted to its 2-ary form (i.e. all leaf nodes have not been added). Therefore, the number of 1's in the code of such a tree is at least equal to the number of zeroes.

[]

It follows from the preceeding lemma that the position of the i^{th} 1 in the code has a position of at most (2i-1). So instead of the code of size 2n, a new code of size n can be created where the i^{th} number in the code gives the position of the i^{th} 1 in the original code. For example, the new code for the above case is (1 2 4 5 8 9).

LEMMA 4.2

The new code has the following properties:

1) The size of the code is n.

2) The value of the i^{th} element of the code is at least i and at most 2i-1 .

3) The elements of the code are in increasing order.

proof:

Since there are n 1's in the original code, the size of the new code is n. Since the original code satisfies the prefix property i.e. the number of 1's in the prefix is never greater than the number of Ø's, the position of the ispth 1 can at most be equal to 2i-1. (3) is obvious from the fact that the position of the i^{th} 1 is greater than the position of the $(i+1)^{th}$ 1.

[]

Let us consider an example and see how the binary trees are generated lexicographically.

EXAMPLE 4.1

Let n = 4.

The minimum tree generated in the lexicographic order is (1 2 3 4) and the maximum is (1 3 5 7). The complete order of generation is as follows:

(1 2 3 4)
(1 2 3 5)
(1 2 3 6)
(1 2 3 7)

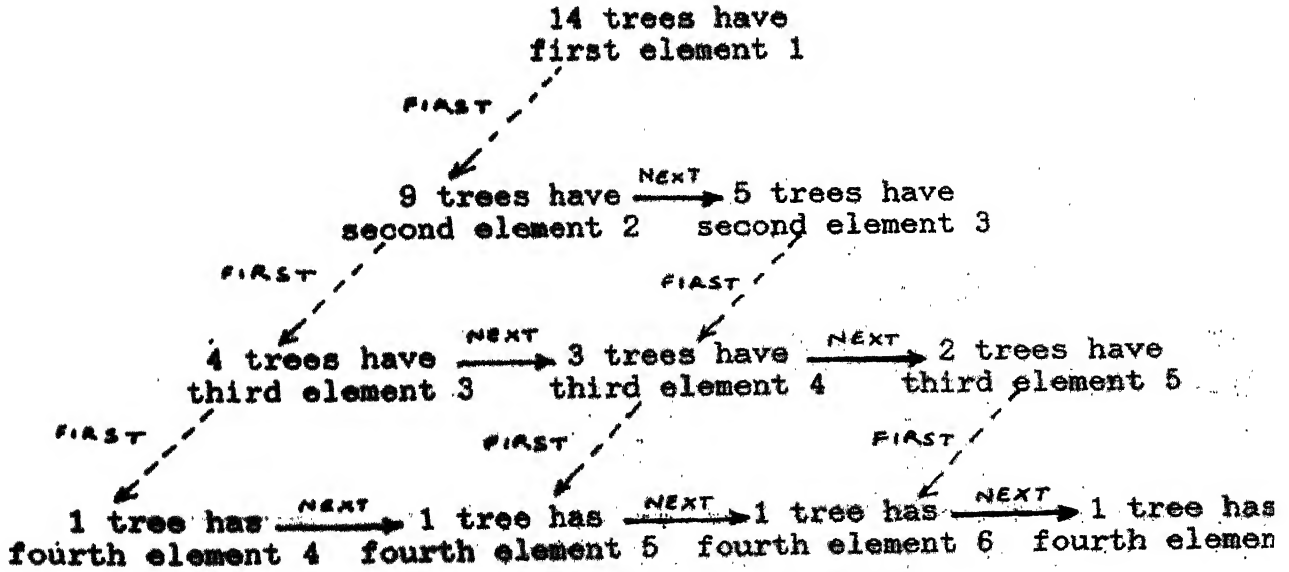
(1 2 4 5)
(1 2 4 6)
(1 2 4 7)
(1 2 5 6)
(1 2 5 7)
(1 3 4 5)
(1 3 4 6)
(1 3 4 7)
(1 3 5 6)
(1 3 5 7)

It can easily be verified that the number of binary trees with n nodes is given by

$$BT(n) = 1/(n+1) \ {}^{2n}C_n \quad \text{..... (ref. [Z8Ø])}$$

In the above example BT(4) = 14.

On careful observation one can clearly see a pattern in the lexicographically ordered codes. In fact, the codes generated in the above example follow the pattern of the following structure:



Let the above structure be stored in an array TRIANGLE[1:n,1:n]. It is easy to see that the elements of TRIANGLE satisfy the following properties:

1) TRIANGLE[i,i] = BT[n-i-1],

2) TRIANGLE[n,i] = 1, $1 \leq i \leq n$,

3) TRIANGLE[i,j] = TRIANGLE[i+1,j] + TRIANGLE[i,j+1], i>j, $1 \leq i \leq n-1$, $1 \leq j \leq n-1$.

In general,

$$\text{TRIANGLE}[i,j] = (i-j+2/n-i)\ {}^{2n-j-i+1}C_{n-i-1}$$

LEMMA 4.3

Each row of TRIANGLE contains partial sums of the elements of the next row.

i.e. $$\text{TRIANGLE}[i,j] = \sum_{k=j}^{i+1} \text{TRIANGLE}[i+1,k]$$

proof:

By construction,

$$\begin{aligned}\text{TRIANGLE}[i,j] &= \text{TRIANGLE}[i+1,j] + \text{TRIANGLE}[i,j+1] \\ &= \text{TRIANGLE}[i+1,j] + \text{TRIANGLE}[i+1,j+1] \\ &\quad + \text{TRIANGLE}[i,j+2] \\ &= \text{TRIANGLE}[i+1,j] + \ldots + \text{TRIANGLE}[i+1,i-1] \\ &\quad + \text{TRIANGLE}[i,i].\end{aligned}$$

Now,

$$\text{TRIANGLE}[i,j] = (i-j+2/n-i)\ {}^{2n-j-i+1}C_{n-i-1}$$

Therefore,

$$\text{TRIANGLE}[i+1,i+1] = (2/n-i-1)\ {}^{2n-2i-1}C_{n-i-2}$$

$$= 2.\ (2n-2i-1)!\ /\ (n-i-1)!.(n-i+1)!$$

$$TRIANGLE[i+1,i] = (3/n-i-1)\ {}^{2n-2i}C_{n-i-2}$$

$$= 3.\ (2n-2i)!\ /\ (n-i-1)!.(n-i+2)!$$

$$TRIANGLE[i+1,i+1] + TRIANGLE[i+1,i]$$

$$= (2/n-i)\ {}^{2n-2i+1}C_{n-i+1}$$

$$= TRIANGLE[i,i]$$

Thus,

$$TRIANGLE[i,j] = \sum_{k=j}^{i+1} TRIANGLE[i+1,k]$$

[]

Each element of the array TRIANGLE defines a set of trees having a a particular integer at a specified position in the code. Thus, after generating the array TRIANGLE, the unranking procedure i.e. generating the tree corresponding to a rank, becomes very obvious. The TRIANGLE is traversed using the FIRST and NEXT pointers to find the sets of trees to which the tree with a given rank belongs to, with the i^{th} row of TRIANGLE contributing the i^{th} element to the code of the tree.

4.3. THE GENERATION ALGORITHMS

The first algorithm, **par_lex_bintree**, generates all the binary trees with n nodes in lexicographic order, one after the other. The algorithm makes use of O(n) processors and O(BT(n)) time; where BT(n) gives the total number of binary trees with n nodes. This is optimal since it takes O(n) time on a serial computer to output one tree. The second algorithm, **par_gen_bintree**, generates all the

binary trees simultaneously in $O(\lceil BT(n)/P \rceil . n)$ time, where P is the number of processors available. This again achieves optimal speedup ratio.

4.3.1 ALGORITHM par_lex_bintree

Let A[1:n] store the code of any binary tree.

The first tree in the lexicographic order is the one in which A[i] = i, for all i. Given a binary tree (code), consider the problem of generation of the next binary tree in the lexicographic order. For this is needed additional storage in the form of an array NEXT[1:n]. NEXT[i] stores the index of the closest element to the left of the i^{th} position in A[1:n] which has not reached its maximum (The maximum element at position i is 2i-1). Thus NEXT[i] = i to start with. Also, the pointer 'pointr' gives the minimum index from which onwards the code of the previous tree has to be changed in order to get the new code. Obviously, the minimum next tree that can be generated is the one in which A[pointr] := A[pointr]+1, and for all i > pointr, A[i] := A[i-1]+1.

After generating the tree the array NEXT and the pointer 'pointr' may be modified for subsequent generations as follows:

If after incrementing, A[pointr] has reached its maximum (i.e. A[pointr] = 2.pointr-1), then if pointr <> n, NEXT[pointr+1] is set to (pointr-1). This is consistent with the definition of the array NEXT. Since none of the elements

with index greater than 'pointr' have reached their maximum, for each of them (except pointr+1) NEXT[i] may be set to i-1. The pointer 'pointr' is set to n since it is the maximum index which has not reached its maximum (Thus it is the minimum index which has to be changed in order to get the next tree). However, if pointr = n, 'pointr' may simply be set to NEXT[pointr].

If, after incrementing, A[pointr] has not reached its maximum, then for all i > pointr, NEXT[i] is set to its initial value i.e. i-1 (since none of the elements beyond 'pointr' have reached their maximum).

LEMMA 4.4

The procedure described above can be used to generate all the BT(n) binary trees with n nodes, lexicographically.

[]

The procedure **par_lex_bintree** implements LEMMA 4.4 .

```
proc  par_lex_bintree  (p,n; n)
var
   /* input */
   n : integer;
   /* output */
   A : array [1..n] of 1..2n-1;
   /* local */
   i,u,count,pointr : integer;
   NEXT : array [1..n] of 1..n
begin
I1: for i := p to p+n-1 pardo
       NEXT[i-p+1] := i-p
    dopar;
```

EXAMPLE 4.2

The following gives the trace of generation of all binary trees with n nodes in lexicographic order using procedure **par_lex_bintree**.

count	A[i]					NEXT[i]					pointer
	i= 1	2	3	4	5	i= 1	2	3	4	5	
1	1	2	3	4	5	Ø	1	2	3	4	5
2	1	2	3	4	6	Ø	1	2	3	4	5
3	1	2	3	4	7	Ø	1	2	3	4	5
4	1	2	3	4	8	Ø	1	2	3	4	5
5	1	2	3	4	9	Ø	1	2	3	4	4
6	1	2	3	5	6	Ø	1	2	3	4	5
7	1	2	3	5	7	Ø	1	2	3	4	5
8	1	2	3	5	8	Ø	1	2	3	4	5
9	1	2	3	5	9	Ø	1	2	3	4	4
10	1	2	3	6	7	Ø	1	2	3	4	5
11	1	2	3	6	8	Ø	1	2	3	4	5
12	1	2	3	6	9	Ø	1	2	3	4	4
13	1	2	3	7	8	Ø	1	2	3	3	5
14	1	2	3	7	9	Ø	1	2	3	3	3
15	1	2	4	5	6	Ø	1	2	3	4	5
16	1	2	4	5	7	Ø	1	2	3	4	5
17	1	2	4	5	8	Ø	1	2	3	4	5
18	1	2	4	5	9	Ø	1	2	3	4	4
19	1	2	4	6	7	Ø	1	2	3	4	5
20	1	2	4	6	8	Ø	1	2	3	4	5
21	1	2	4	6	9	Ø	1	2	3	4	4
22	1	2	4	7	8	Ø	1	2	3	3	5
23	1	2	4	7	9	Ø	1	2	3	3	3
24	1	2	5	6	7	Ø	1	2	2	4	5
25	1	2	5	6	8	Ø	1	2	2	4	5
26	1	2	5	6	9	Ø	1	2	2	4	4
27	1	2	5	7	8	Ø	1	2	2	2	5
28	1	2	5	7	9	Ø	1	2	2	2	2
29	1	3	4	5	6	Ø	1	1	3	4	5
30	1	3	4	5	7	Ø	1	1	3	4	5
31	1	3	4	5	8	Ø	1	1	3	4	5
32	1	3	4	5	9	Ø	1	1	3	4	4
33	1	3	4	6	7	Ø	1	1	3	4	5
34	1	3	4	6	8	Ø	1	1	3	4	5
35	1	3	4	6	9	Ø	1	1	3	4	4
36	1	3	4	7	8	Ø	1	1	3	3	5
37	1	3	4	7	9	Ø	1	1	3	3	3
38	1	3	5	6	7	Ø	1	1	1	4	5
39	1	3	5	6	8	Ø	1	1	1	4	5
40	1	3	5	6	9	Ø	1	1	1	4	4
41	1	3	5	7	8	Ø	1	1	1	1	5
42	1	3	5	7	9	Ø	1	1	1	1	1

4.3.2 ALGORITHM par_gen_bintree

Though the algorithm of last section achieves optimal speedup, it generates all the trees one after the other. Therefore, it is of interest to design an algorithm which can generate trees simultaneously. This is what is accomplished by the algorithm **par_gen_bintree** proposed in this section. The algorithm **par_gen_bintree** generates all the binary trees with n nodes in $O(\lceil BT(n)/P \rceil .n)$ time ,where P is the number of processors used.

The algorithm needs preprocessing in the form of the array TRIANGLE as described in section 4.2. After this, the algorithm unranks all the binary trees simultaneously by traversing TRIANGLE.

The procedure **par_form_tri** computes the array TRIANGLE.

```
proc par_form_tri ( p,n^2; n; TRIANGLE );

var

   /* input */
   n : integer;
   /* output */
   TRIANGLE : array [1..n,1..n] of 1..BT(n);
   /* local */
   i,j,u,v : integer;
   FACT : array [1..2n] of 1..2n!
begin
I1: for i := p to p+2n-1 pardo
       FACT[i-p+1] := i-p+1
    dopar
```

```
I2: for j := Ø to ⌈log 2n⌉ do
       for i:= p to p+2n-1 pardo
          begin
             u := i-p+1;
             FACT[u+2^j] := FACT[u+2^j] * FACT[u]
          end
       dopar;
I3: for i := p to p+n-1 pardo
       for j := p to p+i-1 pardo
          begin
             u := i-p+1;
             v := j-p+1;
             TRIANGLE[u,v] := ((u-v+2)/(n-u)) *
               (FACT[2n-u-v-1]/(FACT[n-u-1]*FACT[n-v-2]))
          end
       dopar
    dopar
end;
```

LEMMA 4.6

The procedure **par_form_tri** computes TRIANGLE in $O(\lceil n^2/P \rceil + \log n)$ time, where P is the number of processors.

proof:

The procedure **par_form_tri** gives the implementation for the case when $P = O(n^2)$. However, given P processors it is trivial to prove that step 2 can be implemented in $O(\lceil n/P \rceil + \log n)$ time. Step I3 of the algorithm takes $O(\lceil n^2/P \rceil)$ time. Hence the overall complexity of the algorithm is $O(\lceil n^2/P \rceil + \log n)$.

[]

The main procedure **par_gen_bintree** is presented next. The BT(n) binary trees generated are stored in BITREES[1:BT(n),1:n].

```
proc  par_gen_bintree  ( p,BT(n); n; BITREES );

var
   /* input */
   n : integer;
   /* output */
   BITREES : array [1..BT(n),1..n] of 1..n;
   /* local */
   RANK : array [1..BT(n)] of 1..BT(n);
   i,j,k,u,v : integer
begin
I1: par_form_tri (p,n^2; n; TRIANGLE);
I2: for i := p to p+BT(n)-1 pardo
       begin
          u := i-p+1;
          RANK[u] := u;
          BITREES[u,1] := 1;
          j := 1;
I3:       for k := p+1 to p+n-1 do
             begin
                v := k-p+1;
                BITREES[u,v] := BITREES[u,v-1] + 1;
                while TRIANGLE[v,j] < RANK[u] do
                   begin
                      RANK[u] := RANK[u] - TRIANGLE[v,j]
                      j := j+1;
                      BITREES[u,v] := BITREES[u,v] + 1
                   end
             end
       end
end;
```

LEMMA 4.7

The procedure **par_gen_bintree** generates the BT(n) binary trees with n nodes in $O(\lceil BT(n)/P \rceil .n)$ time where P is the number of processors.

proof:

Form LEMMA 4.6, step I1 takes $O(\lceil n^2/P \rceil + \log n)$ time. Step I2 is done in parallel for all binary trees. Each binary tree is generated by traversing TRIANGLE such that, if u and v are the two indices of the array, both u and v are nondecreasing. The traversal is carried out in the same fashion as was demonstrated in section 4.2 of this chapter.

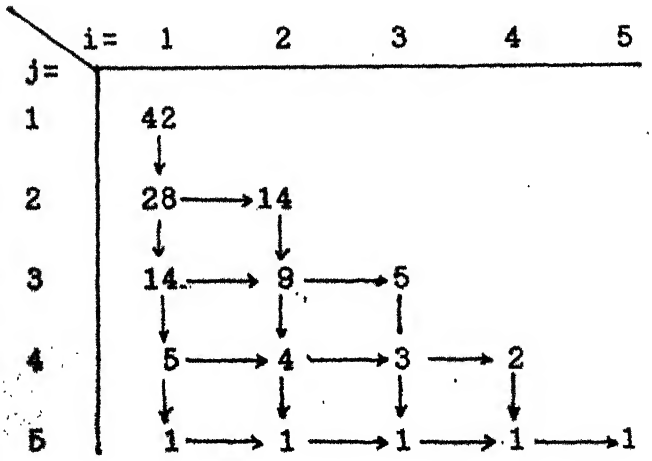
Thus each traversal takes O(n) time. Since P traversals can be done simultaneously, the complexity of step I2 is $O(\lceil BT(n)/P \rceil .n)$. Hence the complexity of the algorithm **par_gen_bintree** is $O(\lceil BT(n)/P \rceil .n)$.

[]

EXAMPLE 4.3

The present example uses procedure **par_gen_bintree** to generate all binary trees with n=5.

The array TRIANGLE computed in step I1 is as follows:



Let us generate the 20^{th} binary tree. The following gives the trace showing how BITREE[20,1:5] is generated.

Initialize,

RANK[20] := 20

BITREES[20,1] := 1

The elements of TRIANGLE are traversed as follows: If the current element being examined TRIANGLE[u,v] is less than RANK[20] then subtract TRIANGLE[u,v] from RANK[20] and

go to TRIANGLE[u,v+1]. Otherwise go to TRIANGLE[u+1,v].

ELEMENT OF TRIANGLE EXAMINED u=	v=	RANK[2Ø]	BITREES[2Ø,u]
2	1	2Ø	②
3	1	6	3
3	2	6	④
4	2	2	5
4	3	2	⑥
5	3	1	7
5	4	1	⑦

Thus the $2Ø^{th}$ binary tree is (1 2 4 6 7).

4.4. CONCLUSION

In this chapter two procedures, **par_lex_bintree** & **par_gen_bintree** have been presented for the algorithmic generation of all binary trees with n nodes, lexicographically, on a parallel random access machine. The procedure **par_lex_bintree** generates all the trees in O(BT(n)) time with O(n) processors. The procedure **par_gen_bintree** uses an unranking procedure to generate all binary trees simultaneously in $O(\lceil BT(n)/P \rceil .n)$ time, P being the number of processors. Both the algorithms are cost optimal. In procedure **par_gen_bintree**, the problem has been parallelised only to the extent that all unrankings are performed simultaneously. However, a quick and easy parallel solution to the unranking problem itself seems improbable. In fact, developing more efficient unranking procedures is an open problem [CDC86].

The procedure **par_lex_bintree** can be generalised to

the case of K-ary trees (i.e. those trees in which no node has more than K sons) by simply replacing 2i-1 by K(i-1) + 1 in step I4

REFERENCES

[AHU74] **Aho A. V.**, Hopcroft J.E., Ullman J.D., *The design and analysis of computer algorithms*, Addison-Wesley, Massachusetts.

[AD85] **Ahrens J.H., Dieter U.**, *Sequential Random Sampling*, ACM Trans. Math. Softw., 1985, 11, 2, 157-169.

[AS85] **Akl S.G.**, *Parallel Sorting Algorithms*, Academic Press, 1985.

[BH82] **Borodin A., Hopcroft J.E.**, *Routing, Merging and Sorting on Parallel Models of Computation*, Proceedings of the *14 sup th* Annual ACM Symposium on Theory of Computing, 1982, 334-344.

[CA86] **Chan B, Akl S.G.** *Generating Combinations in Parallel*, BIT, 1986, 26 ,2-6.

[CC86] **Chen G.H., Chern M.S.**, *Parallel Generation of Permutations and Combinations*, BIT, 1986, 26, 277-283.

[CO86] **Cole R.**, *Parallel Merge Sort*, Proceedings, *27 sup th* Annual symposium on Foundations of Computer Science, 1986, 511-516.

[EN82] **Ernvall J., Nevalainen O.** *An efficient algorithm for unbiased Random Sampling*, Computer Journal,

for unbiased Random Sampling, Computer Journal, 25, 1982, 45-47.

[FMR62] **Fan C.T., Muller M.E., Rezucha I.,** *Development of Sampling plans by using Sequential (item by item) selection techniques on digital computers,* Journal of American Statistical Association, 1962, 57, 387-402.

[FL66] **Flynn M.J.,** *Very High Speed computing system,* Proc. of IEEE, 1966, 54, 1901-1909.

[GB81] **Gupta P., Bhattacharjee G.P.,** *Parallel Generation of Combinations Lexicographically,* FST&TCS 1, 1981, Bangalore, 129-200.

[GB82] **Gupta P., Bhattacharjee G.P.,** *Parallel Generation of Permutations with repetitions Lexicographically,* FST&TCS 2, 1982, Bangalore, 378-399.

[GB83] **Gupta P., Bhattacharjee G.P.,** *Parallel Generation of Permutations,* Computer Journal, 1983, 26, 97-105.

[GB84] **Gupta P., Bhattacharjee G.P.,** *An efficient algorithm for Random Sampling without replacement,* IJCM, 1984, 16, 201-210.

[GH77] **Goodman S.E., Hedetniemi S.T.,** *Introduction to the Design and Analysis of Algorithms,* 1977, McGraw Hill Inc.

[GP83] **Galil Z., Paul W.J.,** *An efficient general purpose*

parallel computer, JACM, vol 30, 360-367.

[GLW82] **Gupta U., Lee D.T., Wong C.K.,** *Ranking and unranking of 2-3 trees,* SIAM J. Comput, vol 11, no 3, 1982, 583-590.

[JO62] **Jones T.G.,** *A note on sampling a tape file,* Commun. ACM, 1962, 5, 6, 343.

[KP81] **Kuhn R.H., Padua D.A.,** *Tutorial on Parallel Processing,* IEEE Comp. Society Press.

[LLW86] **Lee C.C., Lee D.T., Wong C.K.,** *Generating Binary Trees of bounded height,* Acta Informatica, 1986, 23, 529-544.

[KU80] **Kung H.T.,** *The Structure of Parallel Algorithms,* Advances in Computer, 1980, 19, Academic Press, New York.

[K77] **Knott G.D.,** *A numbering system for binary trees,* Commun. ACM, vol 20, no 3, 1977, 113-115.

[K72] **Knuth D.E.,** *The art of computer programming vol 1: Fundamentals of algorithms,* Addition-Wesley, Massachusetts, 1972.

[K73] **Knuth D.E.,** *The art of computer programming vol 2: Seminumerical algorithms,* Addition-Wesley, Massachusetts, 1973.

[MF82] **Mor M., Fraenkel A.S.,** *Permutation Generation on vector processors,* Comp. Journ., 1982, 25, 4, 423-426.

[MS65] **Mayeda W., Sehu S.,** *Generation of trees without duplication,* IEEE-CT, CT-12, 1965,181-185.

[NW75] **Nijenhuis A., Wilf H.S.,** *Combinatorial Algorithms,* Academic Press Inc., 1975.

[PR80] **Pruskurowski A.,** *On generation of Binary Trees,* Journal of ACM, 1980, 27, 1-2.

[QD84] **Quinn M.J., Deo N.,** *Parallel graph algorithms,* Comput. Surveys, vol 16, no 3, 1984, 580-599.

[RND78] **Reingold E.M., Nievergelt J., Deo N.,** *Combinatorial Algorithms,* Prentice Hall, Englewood Cliffs, New Jersey.

[R78] **Ruskey F.,** *Generating t-ary trees lexicographically,* SIAM J. Comput., vol 7, 1978, 424-439.

[SE77] **Sedgewick R.,** *Permutation Generation methods,* ACM Comput. Surverys, 1977, 9, 137-164.

[TN82] **Teuhola J., Nevalainen O.,** *Two efficient algorithms for random sampling without replacement,* IJCM, 1982, 11, 127-140.

[VI87] **Vitter J.S.,** *An efficient algorithm for sequential random sampling,* ACM Trans. Math. Softw., 1987, 13, 1, 58-67.

[VI84] **Vitter J.S.,** *Faster methods for Random Sampling,* Commun. ACM, 1984, 27, 7, 703-718.

[Z80] **Zaks S.,** *Lexicographic generation of ordered trees,* TCS, vol 10, 1980, 63-82.

[ZR79] **Zaks S., Richards D.,** *Generating trees and other combinatorial objects lexicographically,* SIAM J. Comput., vol 7, 1978, 424-439.